Views and Iterators for Generic Constraint Implementations

Christian Schulte[1] Guido Tack[2]

[1] IMIT, KTH - Royal Institute of Technology, Sweden

[2] PS Lab, Saarland University, Saarbrücken, Germany

`schulte@imit.kth.se,tack@ps.uni-sb.de`

Abstract

This paper introduces an architecture for generic constraint implementations based on variable views and range iterators. Views allow, for example, to scale, translate, and negate variables. The paper shows how to make constraint implementations generic and how to reuse a single generic implementation with different views for different constraints. Applications of views exemplify their usefulness and their potential for simplifying constraint implementations. We introduce domain operations compatible with views based on range iterators. The paper evaluates different implementation techniques for the presented architecture.

1 Introduction

A challenging aspect in developing and extending a constraint programming system is implementing a *comprehensive* set of constraints. Ideally, a system should provide simple, expressive, and efficient abstractions that ease development and reuse of constraint implementations.

This paper contributes a new architecture based on variable views and range iterators. The architecture comprises an additional level of abstraction to decouple variable implementations from propagators (as constraint implementations). Propagators compute generically with variable views instead of variables. Views support operations like scaling, translation, and negation of variables. For example, a simple generic propagator for linear equality $\sum_{i=1}^{k} x_i = c$ can be used with a scale-view $x_i = a_i \cdot y_i$ to obtain an implementation of $\sum_{i=1}^{k} a_i \cdot y_i = c$. Variable views assist in implementing propagators on a higher level of abstraction.

Range iterators support powerful and efficient domain operations on variables and variable views. The operations can access and modify multiple values of a variable domain simultaneously. Range iterators are efficient as they help avoiding temporary data structures. They simplify propagators by serving as adaptors between variables and propagator datastructures.

The architecture is carefully separated from its implementation. Two different implementation approaches are presented and evaluated. An implementation using parametric polymorphism (such as templates in C++) is shown to not incur any runtime cost. The architecture can be used for arbitrary constraint programming systems and has been fully implemented in Gecode [2].

Plan of the paper. The next section presents a model for finite domain constraint programming systems. Sect. 3 introduces variable views and exemplifies their use. Sect. 4 introduces iterator-based domain operations that are applied to views in the following section. Variable views for set constraints are discussed in Sect. 6. In Sect. 7 implementation approaches for views and iterators are presented, followed by their evaluation in Sect. 8. The last section concludes.

2 Constraint Programming Systems

This section introduces the model for finite domain constraint programming systems considered in this paper and relates it to existing systems.

Variables and propagators. Finite domain constraint programming systems offer services to support constraint propagation and search. In this paper we are only concerned with variables used for constraint propagation. We assume that a constraint is implemented by a *propagator*. A propagator maintains a collection of variables and performs constraint propagation by executing operations on them. In the following we consider finite domain variables and propagators. A finite domain variable x has an associated *domain* $\mathrm{dom}(x)$ being a subset of some finite subset of the integers.

Propagators do not manipulate variable domains directly but use operations provided by the variable. These operations return information about the domain or update the domain. In addition, they handle failure (the domain becomes empty) and control propagation.

Value operations. A *value operation* on a variable involves a single integer as result or argument. We assume that a variable x with $D = \mathrm{dom}(x)$ provides the following value operations: $x.\mathrm{getmin}()$ returns $\min D$, $x.\mathrm{getmax}()$ returns $\max D$, $x.\mathrm{adjmin}(n)$ updates $\mathrm{dom}(x)$ to $\{m \in D \mid m \geq n\}$, $x.\mathrm{adjmax}(n)$ updates $\mathrm{dom}(x)$ to $\{m \in D \mid m \leq n\}$, and $x.\mathrm{excval}(n)$ updates $\mathrm{dom}(x)$ to $\{m \in D \mid m \neq n\}$. These operations are typical for finite domain constraint programming systems like Choco [6], ILOG Solver [9, 11, 4], Eclipse [1], Mozart [8], and Sicstus [5]. Some systems provide additional operations such as for assigning values.

Domain operations. A *domain operation* supports simultaneous access or update of multiple values of a variable domain. In many systems this is provided by supporting an abstract set-datatype for variable domains, as for example in Choco [6], Eclipse [1], Mozart [8], and Sicstus [5]. ILOG Solver [9, 11, 4] only allows access by iterating over the values of a variable domain.

Subscription. When a propagator p is created, it *subscribes* to its variables. Subscription guarantees that p is executed whenever the domain of one of its variables changes according to an event. An event describes when the propagator requires execution: $e = \mathrm{fix}$ when $\mathrm{dom}(x)$ becomes a singleton, $e = \min$ if $\min \mathrm{dom}(x)$ changes, $e = \max$ if $\max \mathrm{dom}(x)$ changes, and $e = \mathrm{any}$ if $\mathrm{dom}(x)$ changes. For more on events, see for example [13].

Range and value sequences. Range notation $[n \,..\, m]$ is used for the set of integers $\{l \in \mathbb{Z} \mid n \leq l \leq m\}$. A *range sequence* $\mathrm{ranges}(I)$ for a finite integer set $I \subseteq \mathbb{Z}$ is the shortest sequence $s = \langle [n_1 \,..\, m_1], \ldots, [n_k \,..\, m_k] \rangle$ such that I is covered ($\mathrm{set}(s) = I$, where $\mathrm{set}(s)$ is defined as $\bigcup_{i=1}^{k} [n_i \,..\, m_i]$) and the ranges are ordered by their smallest elements ($n_i \leq n_{i+1}$ for $1 \leq i < k$). The above range sequence is also written as $\langle [n_i \,..\, m_i] \rangle_{i=1}^{k}$. Clearly, a range sequence is unique, none of its ranges is empty, and $m_i + 1 < n_{i+1}$ for $1 \leq i < k$.

A *value sequence* $\mathrm{values}(I)$ for a finite set of integers $I \subseteq \mathbb{Z}$ is the shortest sequence $s = \langle n_1, \ldots, n_k \rangle$ (also written as $\langle n_i \rangle_{i=1}^{k}$) such that I is covered ($\mathrm{set}(s) = I$, where $\mathrm{set}(s) = \bigcup_{i=1}^{k} n_i$) and that the values are ordered ($n_i \leq n_{i+1}$ for $1 \leq i < k$). Again, this sequence is unique and has no duplicate values.

3 Variable Views with Value Operations

This section introduces variable views with value operations. The full design with domain operations and a discussion of their properties follows in Sect. 5.

Example 1 (Smart n-Queens) Consider the well-known finite domain constraint model for n-Queens using three alldifferent constraints: each queen is represented by a variable x_i ($0 \leq i < n$) with domain $\{0, \ldots, n-1\}$. The constraints state that the values of all x_i, the values of all $x_i - i$, and the values of all $x_i + i$ must be pairwise different for $0 \leq i < n$.

If the used constraint programming system lacks versions of alldifferent supporting that the values of $x_i + c_i$ are different, the user must resort to using additional variables y_i and constraints $y_i = x_i + c_i$ and the single constraint that the y_i are different. This approach is clearly not very efficient: it triples the number of variables and requires additional $2n$ binary constraints.

Systems with this extension of alldifferent must implement two very similar versions of the same propagator. This is tedious and increases the amount of code that requires maintenance. In the following we make propagators *generic*: the same propagator can be reused for several variants.

To make a propagator generic, all its operations on variables are replaced by operations on variable views. A *variable view* (view for short) implements the same operations as a variable. A view stores a reference to a variable. Invoking an operation on the view executes the appropriate operation on the view's variable. Multiple variants of a propagator can be obtained by instantiating the single generic propagator with multiple different variable views.

Offset-views. For an *offset-view* $v = \text{voffset}(x, c)$ for a variable x and an integer c, performing an operation on v results in performing an operation on $x + c$. The operations on the offset-view are:

$$\begin{array}{llll} v.\text{getmin}() & := x.\text{getmin}() + c & v.\text{getmax}() & := x.\text{getmax}() + c \\ v.\text{adjmin}(n) & := x.\text{adjmin}(n - c) & v.\text{adjmax}(n) & := x.\text{adjmax}(n - c) \\ v.\text{excval}(n) & := x.\text{excval}(n - c) & & \end{array}$$

To obtain both alldifferent propagators required by Example 1, also an *identity-view* is needed. An operation on an identity-view $\text{vid}(x)$ for a variable x performs the same operation on x. That is, identity-views turn variables into views to comply with propagators now computing with views.

Obtaining the two variants of alldifferent is straightforward: the propagator is made generic with respect to which view it uses. Using the propagator with both an identity-view and an offset-view yields the required propagators.

Sect. 7 discusses how views can be implemented whereas this section focuses on the architecture only. However, to give some intuition, in C++ for example, propagators can be made generic by implementing them as templates with the used view as template argument. Instantiating the generic propagator then amounts to instantiating the corresponding template with a particular view.

Views are orthogonal to the propagator. In the above example, offset-views can be used for any implementation of alldifferent using value operations. This includes the naive version propagating when variables become assigned or the bounds-consistent version [10].

Scale-views. In the above example, views allow to reuse the same propagator for variants of a constraint, avoiding duplication of code and effort. In the following, views can also simplify the implementation of propagators.

Example 2 (Linear inequalities) A common constraint is linear inequality $\sum_{i=1}^{n} a_i \cdot x_i \leq c$ (equality and disequality is similar) with integers a_i and c and variables x_i. In the following we restrict the a_i to be positive.

A typical bounds-propagator executes for $1 \leq j \leq n$:

$$x_j.\text{adjmax}(\lceil (c - l_j)/a_j \rceil) \quad \text{with} \quad l_j = \textstyle\sum_{i=1, i \neq j}^{n} a_i \cdot x_i.\text{getmin}()$$

Quite often, models feature the special case $a_i = 1$ for $1 \leq i \leq n$. For this case, it is sufficient to execute for $1 \leq j \leq n$:

$$x_j.\text{adjmax}(c - l_j) \quad \text{with} \quad l_j = \sum_{i=1, i \neq j}^{n} x_i.\text{getmin}()$$

As this case is common, a system should optimize it. An optimized version requires less space (no a_i required) and less time (no multiplication, division, and rounding). But, a more interesting question is: can one just implement the simple propagator and get the full version by using views?

With scale-views, the simple implementation can be used in both cases. A *scale-view* $v = \text{vscale}(a, x)$ for a positive integer $a > 0$ and a variable x defines operations for $a \cdot x$:

$$\begin{array}{ll} v.\text{getmin}() := a \cdot x.\text{getmin}() & v.\text{getmax}() := a \cdot x.\text{getmax}() \\ v.\text{adjmin}(n) := x.\text{adjmin}(\lceil n/a \rceil) & v.\text{adjmax}(n) := x.\text{adjmax}(\lfloor n/a \rfloor) \\ v.\text{excval}(n) := \ \textbf{if } n \bmod a = 0 \textbf{ then } x.\text{excval}(n/a) & \end{array}$$

From the simpler implementation the special case (identity-views) and the general case (scale-views) can be obtained. Multiplication, division, and rounding is separated from actually propagating the inequality constraint. Views hence support separation of concerns and can simplify the implementation of propagators. In particular, multiplication, division, and rounding need to be implemented only once for the scale-view: any generic propagator can use scale-views.

Minus-views. Another common optimization is to implement binary and ternary variants of commonly used constraints. This optimization reduces the overhead with respect to both time and memory as no array is needed.

Example 3 (Binary linear inequality) Consider a propagator for $v_1 + v_2 \leq c$ with views v_1 and v_2 propagating as described in Example 2. With scale-views $v_1 = \text{vscale}(a_1, x_1)$ and $v_2 = \text{vscale}(a_2, x_2)$ the propagator also implements $a_1 \cdot x_1 + a_2 \cdot x_2 \leq c$ provided that $a_1, a_2 > 0$. However, $x_1 - x_2 \leq c$ cannot be obtained with scale-views. Even if scale-views allowed negative constants, it would be inefficient to multiply, divide, and round to just achieve negation.

A *minus-view* $v = \text{vminus}(x)$ for a variable x provides operations such that v behaves as $-x$. Its operations reflect that the smallest possible value for x is the largest possible value for $-x$ and vice versa:

$$\begin{array}{ll} v.\text{getmin}() := -x.\text{getmax}() & v.\text{getmax}() := -x.\text{getmin}() \\ v.\text{adjmin}(n) := x.\text{adjmax}(-n) & v.\text{adjmax}(n) := x.\text{adjmin}(-n) \\ v.\text{excval}(n) := x.\text{excval}(-n) & \end{array}$$

With minus-views, $x_1 - x_2 \leq c$ can be obtained from an implementation of $v_1 + v_2 \leq c$ with $v_1 = \text{vid}(x_1)$ and $v_2 = \text{vminus}(x_2)$. With an offset-view it is actually sufficient to implement $v_1 + v_2 \leq 0$. Then $x_1 + x_2 \leq c$ can be implemented by an identity-view $\text{vid}(x_1)$ for v_1 and an offset-view $\text{voffset}(x_2, -c)$ for v_2. But again, given just $v_1 + v_2 \leq 0$, an implementation for $x_1 - x_2 \leq c$ with $c \neq 0$ cannot be obtained.

Derived views. It is unnecessarily restrictive to define views in terms of variables. The actual requirement for a view is that its variable provides the same operations. It is straightforward to make views generic themselves: views can be defined in terms of other views. The only exception are identity-views as they serve the very purpose of casting a variable into a view. Views such as offset, scale, and minus are called *derived views*: they are derived from some other view.

With derived views being defined in terms of views, the first step to use a derived view is to turn a variable into a view by an identity-view. For example, a minus-view v for the variable x is obtained from a minus-view and an identity-view: $v = \text{vminus}(\text{vid}(x))$.

Example 4 (Binary linear inequality reconsidered) With the help of offset-views, minus-views, and scale-views, all possible variants of binary linear inequalities can now be obtained from a propagator for $v_1 + v_2 \leq 0$. For example, $a \cdot x_1 - x_2 \leq c$ with $a > 0$ can be obtained with $v_1 = \text{vscale}(a, \text{vid}(x_1))$ and $v_2 = \text{vminus}(\text{voffset}(\text{vid}(x_2), c))$ or $v_2 = \text{voffset}(\text{vminus}(\text{vid}(x_2)), -c)$.

Scale-views reconsidered. The coefficient of a scale-view is restricted to be positive. Allowing arbitrary non-zero constants a in a scale-view $s = \text{vscale}(a, x)$ requires to take the signedness of a into account. This can be seen for the following two operations (the others are similar):

$$\begin{aligned} s.\text{getmin}() &:= \textbf{if } a < 0 \textbf{ then } a \cdot x.\text{getmax}() \textbf{ else } a \cdot x.\text{getmin}() \\ s.\text{adjmax}(n) &:= \textbf{if } a < 0 \textbf{ then } x.\text{adjmin}(\lfloor n/a \rfloor) \textbf{ else } x.\text{adjmax}(\lfloor n/a \rfloor) \end{aligned}$$

This extension might be inefficient. Consider Example 2: inside the loop implementing propagation on all views, the decision whether the coefficient in question is positive or negative must be made. For modern computers, conditionals — in particular in tight loops — can reduce performance considerably. A more efficient way is to restrict scale-views to positive coefficients and use an additional minus-view for cases where negative coefficients are required.

Example 5 (Linear inequalities reconsidered) An efficient way to implement a propagator for linear inequality distinguishes positive and negative variables as in $\sum_{i=1}^{n} x_i + \sum_{i=1}^{m} -y_i \leq c$.

The propagator is simple: it consists of two parts, one for the x_i and one for the y_i. Both parts share the same implementation used with different views. To propagate to the x_i, identity-views are used. To propagate to the y_i, minus-views are used. Arbitrary coefficients are obtained from scale-views as shown above.

The example shows that it can be useful to make parts of a propagator generic and reuse these parts with different views. Puget presents in [10] an algorithm for the bounds-consistent alldifferent. The paper presents only an algorithm for adjusting the upper bounds of the variables x_i and states that the lower bounds can be adjusted by using the same algorithm on variables y_i where $y_i = -x_i$. With views, this technique for simplifying the presentation of an algorithm readily carries over to its implementation: the implementation can be reused together with minus-views.

Another aspect in the design of scale-views is numerical precision: for large a the values occurring in computations with a scale-view $\text{vscale}(a, v)$ can exceed the range of normal integer computations. A system can provide several scale-views that differ in which numerical datatype is used for computation. For example, a system can provide a scale-view that computes with normal integers or with double-precision floating point numbers.

Constant-views. Derived views exploit that views do not need to be implemented in terms of variables. This can be taken to the extreme in that a view has no access at all to a variable. A constant-view $v = \text{vcon}(c)$ for an integer c provides operations such that v behaves as a variable x being equal to c:

$$\begin{aligned} v.\text{getmin}() &:= c & v.\text{getmax}() &:= c \\ v.\text{adjmin}(n) &:= \textbf{if } n > c \textbf{ then fail} & v.\text{adjmax}(n) &:= \textbf{if } n < c \textbf{ then fail} \\ v.\text{excval}(n) &:= \textbf{if } n = c \textbf{ then fail} \end{aligned}$$

Example 6 (Ternary linear inequalities) Another optimization for linear constraints are ternary variants. Given a propagator for $v_1 + v_2 + v_3 \leq c$ and using a constant-view $\text{vcon}(0)$ for one of the views v_i, all binary variants as discussed earlier can be obtained.

In summary, for linear inequalities (this carries over to linear equalities and disequalities), views support many optimized special cases from just two implementations (the general n-ary case and the ternary case). These implementations are simple as they do not need to consider coefficients.

Events. The handling of events in subscribing to a variable is straightforward. Clearly, for a constant-view subscription does nothing. For all but the minus-view the events on the variable and the events on the derived view coincide. If $v' = \text{vminus}(v)$, then a change of the minimum of v corresponds to a change of the maximum of v' and vice versa.

4 Domain Operations and Range Iterators

Today's constraint programming systems support domain operations either only for access or by means of an explicitly represented abstract datatype. In this paper, we propose domain operations based on range iterators. These operations are shown to be simple, expressive, and efficient. Additionally, range iterators are essential for views as presented in Sect. 5.

Range iterators. A *range iterator* r for a range sequence $s = \langle [n_i \mathinner{..} m_i] \rangle_{i=1}^{k}$ allows to iterate over s: each of the $[n_i \mathinner{..} m_i]$ can be obtained in sequential order but only one at a time. A range iterator r provides the following operations: $r.\text{done}()$ tests whether all ranges have been iterated, $r.\text{next}()$ moves to the next range, and $r.\text{min}()$ and $r.\text{max}()$ return the minimum and maximum value for the current range. By $\text{set}(r)$ we refer to the set defined by an iterator r (which must coincide with $\text{set}(s)$).

A possible implementation of a range iterator r for s maintains an index i_r which is initially $i_r = 1$, the operations can then be defined as:

$$\begin{array}{ll} r.\text{done}() := i_r > k & r.\text{next}() := (i_r \leftarrow i_r + 1) \\ r.\text{min}() := n_{i_r} & r.\text{max}() := m_{i_r} \end{array}$$

A range iterator hides its implementation. Iteration can be by position as above, but it can also be by traversing a list. The latter is particularly interesting if variable domains are implemented as lists of ranges themselves.

Iterators are consumed by iteration. Hence, if the same sequence needs to be iterated twice, a fresh iterator is needed. If iteration is cheap, a reset-operation for an iterator can be provided so that multiple iterations are supported by the same iterator. For more expensive iterators, a solution is discussed later.

Domain operations. Variables are extended with operations to access and modify their domains with range iterators. For a variable x, the operation $x.\text{getdom}()$ returns a range iterator for $\text{ranges}(\text{dom}(x))$. For a range iterator r the operation $x.\text{setdom}(r)$ updates $\text{dom}(x)$ to $\text{set}(r)$ provided that $\text{set}(r) \subseteq \text{dom}(x)$. The responsibility for ensuring that $\text{set}(r) \subseteq \text{dom}(x)$ is left to the programmer and hence requires careful consideration. Later richer (and safe) domain operations are introduced. The operation $x.\text{setdom}(r)$ is *generic* with respect to r: any range iterator can be used.

Domain operations can offer a substantial improvement over value operations, if many values need to be removed from a variable domain simultaneously. Assume a typical implementation of a variable domain D which organizes $\text{ranges}(D) = \langle [n_i \mathinner{..} m_i] \rangle_{i=1}^{k}$ as a linked-list. Removing a single element from D takes $O(k)$ time and might increase the length of the linked-list by one (introducing an additional hole). Hence, in the worst case, removing l elements takes $O(l(k+l))$ time. With domain operations based on iterators, removal takes $O(k+l)$ time.

Range iterators serve as simplistic abstract datatype to describe finite sets of integers. However, they provide some essential advantages over an explicit set representation. First, any range iterator regardless of its implementation can be used to update the domain of a variable. This turns out to allow for simple, efficient, and expressive updates of variable domains. Second, no costly memory management is required to maintain a range iterator as it provides access to only one range at a time. Third, iterators are essential in providing domain operations on variable views as will be discussed in Sect. 5.

Intersection iterators. Let us consider intersection as an example for computing with range iterators. Intersection is computed by an intersection iterator r (returned by $\mathrm{iinter}(a,b)$) taking two range iterators a and b as input where $\mathrm{set}(r) = \mathrm{set}(a) \cap \mathrm{set}(b)$. The intersection iterator maintains integers n and m for storing the smallest and largest value of its current range. When initialized, the next operation is executed once. The operations for access are obvious: $r.\mathrm{min}()$ returns n and $r.\mathrm{max}()$ returns m. The other operations are:

```
$r.\mathrm{done}() := a.\mathrm{done}() \vee b.\mathrm{done}()$
$r.\mathrm{next}() :=$ if $a.\mathrm{done}() \vee b.\mathrm{done}()$ then return
         repeat
             while $\neg a.\mathrm{done}() \wedge (a.\mathrm{max}() < b.\mathrm{min}())$ do $a.\mathrm{next}()$
             if $a.\mathrm{done}()$ then return
             while $\neg b.\mathrm{done}() \wedge (b.\mathrm{max}() < a.\mathrm{min}())$ do $b.\mathrm{next}()$
             if $b.\mathrm{done}()$ then return
         until $a.\mathrm{max}() \geq b.\mathrm{min}()$
         $n \leftarrow \max(a.\mathrm{min}(), b.\mathrm{min}())$; $m \leftarrow \min(a.\mathrm{max}(), b.\mathrm{max}())$
         if $a.\mathrm{max}() < b.\mathrm{max}()$ then $a.\mathrm{next}()$ else $b.\mathrm{next}()$
```

The **repeat**-loop iterates a and b until their ranges overlap. The tests whether a or b are done ensure that no operation is performed on a done iterator. The remainder computes the resulting range and prepares for computing a next range.

The iterators a and b can be arbitrary iterators (again, the intersection iterator is *generic*), so it is easy to obtain an iterator that computes the intersection of three iterators by using two intersection iterators. Intersection is but one example for a generic iterator, other useful iterators are for example: $r = \mathrm{iunion}(a,b)$ for iterating the union of a and b ($\mathrm{set}(r) = \mathrm{set}(a) \cup \mathrm{set}(b)$) and $r = \mathrm{iminus}(a,b)$ for iterating the set difference of a and b ($\mathrm{set}(r) = \mathrm{set}(a) \setminus \mathrm{set}(b)$).

Example 7 (Propagating equality) Consider a propagator that implements domain-consistent equality: $x = y$ (assuming that x and y are variables, views are discussed later). The propagator can be implemented as follows: get range iterators for x and y by $rx = x.\mathrm{getdom}()$ and $ry = y.\mathrm{getdom}()$, create an intersection iterator $ri = \mathrm{iinter}(rx, ry)$, update one of the variable domains by $x.\mathrm{setdom}(ri)$, and copy the domain from x to y by $y.\mathrm{setdom}(x.\mathrm{getdom}())$.

Cache-iterators. The above example suggests that for some propagators it is better to actually create an intermediate representation of the range sequence computed by an iterator. The intermediate representation can be reused as often as needed. This is achieved by a *cache-iterator*: it takes an arbitrary range iterator as input, iterates it completely, and stores the obtained ranges in an array. Its actual operations then use the array. The cache-iterator also implements a reset operation as discussed above. By this, the possibly costly input iterator is used only once, while the cache-iterator can be used as often as needed.

Richer domain operations. With the help of iterators, richer domain operations are effortless. For a variable x and a range iterator r, the operation $x.\mathrm{adjdom}(r)$ replaces $\mathrm{dom}(x)$ by $\mathrm{dom}(x) \cap \mathrm{set}(r)$, whereas $x.\mathrm{excdom}(r)$ replaces $\mathrm{dom}(x)$ by $\mathrm{dom}(x) \setminus \mathrm{set}(r)$:

$$\begin{aligned} x.\mathrm{adjdom}(r) &:= x.\mathrm{setdom}(\mathrm{iinter}(x.\mathrm{getdom}(), r)) \\ x.\mathrm{excdom}(r) &:= x.\mathrm{setdom}(\mathrm{iminus}(x.\mathrm{getdom}(), r)) \end{aligned}$$

Value versus range iterators. Another design choice is to base domain operations on value iterators: iterate values rather than ranges of a set. This is not efficient: a value sequence is considerably longer than a range sequence (in particular for the common case of a singleton range sequence).

For implementing propagators, however, it can be simpler to iterate values. This can be achieved by a range-to-value iterator. A value iterator v has the operations $v.\mathrm{done}()$, $v.\mathrm{next}()$, and $v.\mathrm{val}()$ to access the current value. A range-to-value iterator takes a range iterator as input and returns a value iterator iterating the values of the range sequence. The inverse is a value-to-range iterator: it takes as input a value iterator and returns the corresponding range iterator. For a value-to-range iterator it is helpful to allow duplicates in the increasing value sequence. This iterator has many applications, as is explained below.

Iterators as adaptors. Global constraints are typically implemented by a propagator computing over some involved data structure, such as for example a variable-value graph for domain-consistent all-distinct [12]. After propagation, the new variable domains must be transferred from the data structure to the variables. This can be achieved by using a range or value iterator as adaptor. The adaptor operates on the data structure and iterates the value or range sequence for a particular variable. The iterator (together with a value-to-range iterator for a value iterator) then can be passed to the appropriate domain operation.

5 Variable Views with Domain Operations

This section discusses domain operations for variable views using iterators.

Identity and constant views. Domain operations for identity-views and constant-views are straightforward. The domain operations for an identity-view $v = \mathrm{vid}(x)$ use the domain operations on x: $v.\mathrm{getdom}() := x.\mathrm{getdom}()$ and $v.\mathrm{setdom}(r) := x.\mathrm{setdom}(r)$. For a constant-view $v = \mathrm{vcon}(c)$, the operation $v.\mathrm{getdom}()$ returns an iterator for the singleton range sequence $\langle [c \mathinner{..} c] \rangle$. The operation $v.\mathrm{setdom}(r)$ just checks whether the range sequence of r is empty.

Derived views. Domain operations for an offset-view $\mathrm{voffset}(v, c)$ are provided by an offset-iterator. The operations of an offset-iterator o for a range iterator r and an integer c (created by $\mathrm{ioffset}(r, c)$) are as follows:

$$\begin{array}{ll} o.\mathrm{min}() \;:= r.\mathrm{min}() + c & \quad o.\mathrm{max}() := r.\mathrm{max}() + c \\ o.\mathrm{done}() := r.\mathrm{done}() & \quad o.\mathrm{next}() := r.\mathrm{next}() \end{array}$$

The domain operations for an offset view $v = \mathrm{voffset}(x, c)$ are as follows:

$$v.\mathrm{getdom}() := \mathrm{ioffset}(x.\mathrm{getdom}(), c) \qquad v.\mathrm{setdom}(r) := x.\mathrm{setdom}(\mathrm{ioffset}(r, -c))$$

For minus-views we just give the range sequence as iteration is obvious. For a given range sequence $\langle [n_i \mathinner{..} m_i] \rangle_{i=1}^k$, the negative sequence is obtained by reversal and sign change as $\langle [-m_{k-i+1} \mathinner{..} -n_{k-i+1}] \rangle_{i=1}^k$. The same iterator for this sequence can be used both for setdom and getdom operations. Note that the iterator is quite complicated as it changes direction of the range sequence, possible implementations are discussed in Sect. 7.

Assume a scale-view $s = \mathrm{vscale}(a, v)$ with $a > 0$ and $\langle [n_i \mathinner{..} m_i] \rangle_{i=1}^k$ being a range sequence for v. If $a = 1$, the range sequence remains unchanged. Otherwise, the corresponding range sequence for s is $\langle \{a \cdot n_1\}, \{a \cdot (n_1 + 1)\}, \ldots, \{a \cdot m_1\}, \ldots, \{a \cdot n_k\}, \{a \cdot (n_k + 1)\}, \ldots, \{a \cdot m_k\} \rangle$.

Assume that $\langle [n_i \mathinner{..} m_i] \rangle_{i=1}^k$ is a range sequence for s. Then the ranges $[\lceil n_i/a \rceil \mathinner{..} \lfloor m_i/a \rfloor]$ for $1 \leq i \leq k$ correspond to the required variable domain for v, however they do not necessarily form a range sequence as the ranges might be empty, overlapping, or adjacent. Iterating the range sequence is simple by skipping empty ranges and conjoining overlapping or adjacent ranges.

Consistency. An important issue is how views affect the consistency of a propagator. Let us first consider all views except scale-views. These views compute bijections on the values as well as on the ranges of a domain D. A bounds (domain) consistent propagator for a constraint C with variables $x_1, \dots, x_n$ establishes bounds (domain) consistency for the constraint C with all the variables replaced by $v_k(x_k)$ (if v_k computes the view of x_k).

Scale-views only compute bijections on values: a range does not remain a range after multiplication. This implies that bounds consistent propagators do not establish bounds consistency on scale-views. Consider for example a bounds consistent propagator for alldifferent. With $x, y, z \in \{1, 2\}$, alldifferent$(4x, 4y, 4z)$ cannot detect failure, while alldifferent(x, y, z) can. Note that this is not a limitation of our approach but a property of multiplication.

6 Views for Set Constraints

Views and iterators readily carry over to other constraint domains. This section shows how to apply them to finite sets.

Finite sets. Most systems approximate domains of finite set variables by a greatest lower and least upper bound [3]: $\mathrm{dom}(x) = (\mathrm{glb}, \mathrm{lub})$. The fundamental operations are similar to domain operations on finite domain variables: $x.\mathrm{glb}()$ returns $\mathrm{glb}(x)$, $x.\mathrm{lub}()$ returns $\mathrm{lub}(x)$, $x.\mathrm{adjglb}(D)$ updates $\mathrm{dom}(x)$ to $(\mathrm{glb}(x) \cup D, \mathrm{lub}(x))$, and $x.\mathrm{adjlub}(D)$ updates $\mathrm{dom}(x)$ to $(\mathrm{glb}(x), \mathrm{lub}(x) \cap D)$.

All these operations take sets as arguments, so iterators play an important role here. In fact, range iterators provide exactly the operations that set propagators need: union, intersection, and complement. Most propagators thus do not require temporary datastructures.

As before, propagators now operate on views. In addition to the identity view, the following derived views make propagators more generic. As for finite domains, *constant-views* – like the empty set, the universe, or some arbitrary set – help derive binary propagators from ternary ones. For example, $s_1 \cap s_2 = s_3$ implements set disjointness if s_3 is the constant empty set. With a *complement-view*, $s_1 = s_2 \setminus s_3$ can be implemented as $s_1 = s_2 \cap \overline{s_3}$

Cross-domain views. With finite domain and set constraints in a single system, cross-domain views come into play. The most obvious cross-domain view is a finite domain variable viewed as singleton set. Using generic propagators, this immediately leads to domain-connecting constraints like $x \in s$ (using $\{x\} \subseteq s$), or $s = \{x_1, \dots, x_k\}$ (using $s = \{x_1\} \uplus \cdots \uplus \{x_k\}$).

For cross-domain views, variable subscription handles the different sets of events. For instance, if a finite domain variable x has an any event, all propagators subscribed to upper-bound-change events of a singleton-view of x must be executed.

Cross-domain views can support more than one implementation for the same variable type. Set variables, for example, can be implemented with lower and upper bounds or with their full domain using ROBDDs [7]. A cross-domain view allows lower/upper bound propagators to operate on ROBDD-based sets, reusing propagators for which no efficient BDD representation exists.

7 Implementation

The presented architecture can be implemented as an orthogonal layer of abstraction for any constraint programming system. This section presents the fundamental mechanisms necessary for iterators and views.

Polymorphism. The implementation of generic propagators, views, and iterators requires *polymorphism*: propagators operate on different views, domain operations and iterators on different iterators. Both subtype polymorphism (through inheritance in Java, inheritance and virtual methods in C++) and parametric polymorphism (through templates in C++, generics in Java, polymorphic functions in ML or Haskell) can be used.

In C++, parametric polymorphism through templates is resolved at compile-time, and the generated code is monomorphic. This enables the compiler to perform aggressive optimizations, in particular inlining. The hope is that the additional layer of abstraction can be optimized away entirely. Some ML compilers also apply monomorphization, so similar results could be achieved. Java generics are compiled into casts and virtual method calls, probably not gaining much efficiency.

Achieving high efficiency in C++ with templates sacrifices expressiveness. Instantiation can *only* happen at compile-time. Hence, either C++ must be used for modeling, or all potentially required propagator variants must be provided by explicit instantiation. For *n*-ary constraints like linear inequality, this can be a real limitation, as all arrays must be monomorphic (only a single kind of view per array is allowed). The same holds true for *n*-ary set constraints, where especially constant-views for the empty set or the universe would come in handy.

A compromise is to use template-based polymorphism whenever possible and only resort to subtype polymorphism when necessary. This can be achieved by instantiating all propagators with special views that can be subclassed, and having an operation to construct such a view from an identity view. In Gecode, we currently only use template-based polymorphism. This makes the system very efficient, as will be analyzed in the next section.

For the instantiation of templates as well as for inlining, the code that should be instantiated or inlined must be available at compile time of the code that uses it. This is why most of the actual code in Gecode resides in C++ header files, slowing down compilation of the system. On the interface level however, no templates are used, such that the header files needed for *using* the library are reasonably small.

System requirements. Variable views and range iterators can be added as an orthogonal extension to existing systems. While value operations are not critical as discussed in Sect. 2, depending on which domain operations a system provides, efficiency can differ. In the worst case, domain operations need to be translated into value operations. This would decrease efficiency considerably, however intermediate computations on range iterators would still be carried out efficiently.

A particularly challenging aspect is reversal of range sequences required for the minus-iterator. One approach to implement reversal is to extend all iterators such that they can iterate both backwards and forwards. Another approach is similar to a cache-iterator: store the ranges generated from the input iterator in an array and iterate in reverse order from the array. In Gecode, we have chosen so far the latter approach due to its simplicity. We are going to explore also the former approach: as variable domains in Gecode are provided as doubly-linked lists, iteration in both directions can be provided efficiently.

8 Analysis and Evaluation

This section analyzes the impact different implementations of iterators and views have on efficiency. Two aspects are evaluated: compile-time polymorphism versus run-time polymorphism, and iterators versus temporary data structures.

The experiments use the Gecode C++ constraint programming library [2]. All tests were carried out on a Intel Pentium IV with 2.8GHz and 1GB of RAM, running Linux. Runtimes are the average of 25 runs, with a coefficient of deviation less than 2.5% for all benchmarks. The *optimized* column gives the time

Table 1: Runtime comparison

Benchmark	optimized	virtual	temporary
	time in ms	relative %	
Alpha	90.50	147.20	103.00
Donald	0.79	124.10	100.00
Golomb 10 (bound)	314.00	173.20	101.70
Golomb 10 (domain)	512.80	138.40	100.50
Magic Sequence 500	270.60	173.50	102.80
Magic Square 6	0.96	149.00	102.30
Partition 32	4 881.20	179.70	102.40
Photo	116.84	141.50	103.50
Queens 100	1.75	146.90	100.00
Crew	5.20	—	159.40
Golf 8-4-9	356.00	—	196.90
Hamming 20-3-32	1 539.60	—	163.00
Steiner 9	110.80	—	226.10

in milliseconds of the optimized system, the other columns are relative to *optimized*. The examples used are standard benchmarks, the first group using only integer constraints, the second group using mainly set constraints.

Code inspection. A thorough inspection of the code generated by several C++ compilers shows that they actually perform the optimizations we consider essential. Operations on both views and iterators are inlined entirely and thus implemented in the most efficient way. The abstractions do not impose a runtime penalty (compared to a system without views and iterators).

Templates versus virtual methods. As the previous section suggested, in C++, compile-time polymorphism using templates is far more efficient than virtual method calls. To evaluate this, we changed the basic operations of finite domain views into virtual methods. The required changes are rather involved, so we did not try the same for iterators and set views. The numbers give an idea of how much more efficient a template-based implementation is. Table 1 shows the results in column *virtual*. Virtual method calls cause a runtime overhead between 25% and 80%.

Temporary datastructures. One important claim is that iterators are advantageous because they avoid temporary datastructures. Table 1 shows in column *temporary* that computing temporary datastructures has limited impact on finite domain variables (about 3%), but considerable impact for set constraints (59% to 126% overhead). Temporary datastructures have been emulated by wrapping all iterators in a cache-iterator as described in Sect. 4.

9 Conclusion

The paper has introduced an architecture decoupling propagators from variables based on views and range iterators. We have argued how to make propagators generic, simpler, and reusable with views for different constraints. We have introduced range iterators as abstractions for efficient domain operations compatible with views. The architecture has been shown to be applicable to finite domain and finite set constraints. Using parametric polymorphism for views and iterators leads to an efficient implementation that incurs no runtime cost.

Acknowledgements

Christian Schulte is partially funded by the Swedish Research Council (VR) under grant 621-2004-4953. Guido Tack is partially funded by DAAD travel grant D/05/26003. Thanks to Patrick Pekczynski for help with the benchmarks, and to Mikael Lagerkvist and the anonymous reviewers for helpful comments.

References

[1] P. Brisset, H. El Sakkout, T. Frühwirth, W. Harvey, M. Meier, S. Novello, T. Le Provost, J. Schimpf, and M. Wallace. ECLiPSe Constraint Library Manual 5.8. User manual, IC Parc, London, UK, Feb. 2005.

[2] Gecode: Generic constraint development environment, 2005. Available upon request from the authors, `www.gecode.org`.

[3] C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.

[4] ILOG S.A. *ILOG Solver 5.0: Reference Manual*. Gentilly, France, Aug. 2000.

[5] I. S. Laboratory. SICStus Prolog user's manual, 3.12.1. Technical report, Swedish Institute of Computer Science, Box 1263, 164 29 Kista, Sweden, Apr. 2005.

[6] F. Laburthe. CHOCO: implementing a CP kernel. In N. Beldiceanu, W. Harvey, M. Henz, F. Laburthe, E. Monfroy, T. Müller, L. Perron, and C. Schulte, editors, *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, number TRA9/00, pages 71–85, 55 Science Drive 2, Singapore 117599, Sept. 2000.

[7] V. Lagoon and P. J. Stuckey. Set domain propagation using ROBDDs. In Wallace [14], pages 347–361.

[8] T. Müller. *Propagator-based Constraint Solving*. Doctoral dissertation, Universität des Saarlandes, Fakultät für Mathematik und Informatik, Fachrichtung Informatik, Im Stadtwald, 66041 Saarbrücken, Germany, 2001.

[9] J.-F. Puget. A C++ implementation of CLP. In *Proceedings of the Second Singapore International Conference on Intelligent Systems (SPICIS)*, pages B256–B261, Singapore, Nov. 1994.

[10] J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 359–366, Madison, WI, USA, July 1998. AAAI Press/The MIT Press.

[11] J.-F. Puget and M. Leconte. Beyond the glass box: Constraints as objects. In J. Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 513–527, Portland, OR, USA, Dec. 1995. The MIT Press.

[12] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 362–367, Seattle, WA, USA, 1994. AAAI Press.

[13] C. Schulte and P. J. Stuckey. Speeding up constraint propagation. In Wallace [14], pages 619–633.

[14] M. Wallace, editor. *Tenth International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*. Springer-Verlag, Toronto, Canada, Sept. 2004.